Live-UMLRT: A Tool for Live Modeling of UML-RT Models

Mojtaba Bagherzadeh
Queen's University
mojtaba@cs.queensu.ca

Karim Jahed
Queen's University
jahed@cs.queensu.ca

Benoit Combemale
University of Toulouse
benoit.combemale@irit.fr

Juergen Dingel
Queen's University
dingel@cs.queensu.ca

***Abstract*—In the context of Model-driven Development (MDD) models can be executed by interpretation or by the translation of models into existing programming languages, often by code generation. This work presents *Live-UMLRT*, a tool that supports live modeling of UML-RT models when they are executed by code generation. *Live-UMLRT* is entirely independent of any live programming support offered by the target language. This independence is achieved with the help of a model transformation which equips the model with support for, e.g., debugging and state transfer both of which are required for live modeling. A subsequent code generation then produces a self-reflective program that allows changes to the model elements at runtime (through synchronization of design and runtime models). We have evaluated *Live-UMLRT* on several use cases. The evaluation shows that (1) code generation, transformation, and state transfer can be carried out with reasonable performance, and (2) our approach can apply model changes to the running execution faster than the standard approach that depends on the live programming support of the target language.**

A demonstration video: https://youtu.be/6GrR-Y9je7Y

***Index Terms*—MDD, Live Modeling, Model Execution, Code Generation**

I. Introduction

Thanks to existing MDD tools many facilities are available to simplify software development using models specifically in the domain of Real-time Embedded Systems (RTE). One of the main facilities is the execution of models, which is supported either by interpretation (sometimes also referred to as simulation) or by the translation of models into existing programming languages, often by code generation (translational execution) [1].

Live programming [2] aims to free developers from the "edit-compile-run" cycle, and allows them to change programs at runtime and get immediate feedback on the change. Often, a form of live programming is supported by several existing programming languages and Integrated Development Environments (IDEs) (e.g., [3]), and its benefits and utility are discussed in several studies (e.g., [4], [5]). Inspired by this line of work, some efforts [6]–[8] have recently been made towards live modeling, i.e., the support for changes to the models while they are being executed. However, this work has focused only on model interpretation, and no work supports live modeling when the models are executed by code generation into general programming languages (GPL).

As suggested by [8], a possible solution for supporting live modeling in the context of translational execution is implementing the live modeling features on top of the target language of the source code being generated. This approach appears straight-forward, but has several problems including: (1) **Edit latency.** The typical sequence of steps for reflecting changes to a model to its running execution consists of: (a) incremental code generation, (b) compile & built, and (c) applying the update to the execution. Even for a small model, these three steps together typically take more than half a second. Any delay of more than 500ms in this context is considered harmful to user experience, and can decrease developer productivity [9], [10]. Increasing model size is likely to increase the delay and exacerbate the problem. (2) **Dependency on the target language of the generated code.** Different programming languages provide different levels of support for live programming. So, the support for live modeling may be limited by the capabilities of the target language.

This work presents *Live-UMLRT*, a tool that supports live modeling of UML-RT (a language for the modeling of soft real-time systems [11]) when they are executed by code generation. *Live-UMLRT* is entirely independent of any live programming support offered by the target language. This independence is achieved with the help of model transformation [12] and code generation techniques. The implementation of *Live-UMLRT* consists of two phases: (1) *Generation of a Self-reflective Program* which is realized through: (a) automatic instrumentation and refinement of models using model transformation techniques to allow for the saving and restoring of previous execution state, which is necessary to support re-execution and the transfer of program state, (b) generation of reflective target code that allows not only introspection of the program execution at runtime, but also changes to the model elements (through a synchronization of design and runtime models), and (c) creation of a debugger plugin that hooks into the execution of the model, and uses the self-reflective features of the generated code to provide live modeling services. (2) *Live modeling using the self-reflective program* which is realized via the direct interaction with the self-reflective program. This decreases the edit latency significantly since there is no need for code generation, compile & build, and hot-swapping for each edit. Also, model debugging services provide an infrastructure for live feedback and safe state transfer.

Live-UMLRT provides a full set of services for live modeling of UML-RT state machines, such as an update mechanism

that prevents inconsistent execution states, adding/removing states and transitions, and adding/removing action code. We have evaluated the prototype on several use cases. The experimental evaluation shows that (1) code generation and refinement can be carried out with reasonable performance, and (2) our approach can apply model changes to the running execution much faster than the standard approach that depends on the live programming support of the target language (i.e., minimize edit latency).

II. Background

A. UML-RT

UML for Real-Time (UML-RT) [11], [13] is a modeling language for designing Real-Time Embedded (RTE) systems with soft real-time constraints. It has been the basis of a lot of academic work, industrial projects, and successful tools (e.g., IBM RSA-RTE, HCL RTist, Eclipse eTrice, and Papyrus-RT [14]). In UML-RT, a system is designed as a set of interacting capsules. A *capsule* is an active object which has *attributes*, *operations*, *ports*, and a *behaviour* modelled by means of a hierarchical state machine [11]. A *protocol* defines the different incoming and outgoing *messages* a capsule can receive or send through its ports. A port is the sole interface for the communication between the capsules which guarantees high encapsulation. Ports of two capsules can be connected through *connectors* only if they are typed with the same protocol.

A UML-RT State Machine (USM) consists of several *states* connected with *transitions*. States can be of three kinds: basic states, composite states (containing sub-states), and pseudo-states (e.g., initial pseudo-state, choice point). A basic or composite state can have *entry* and *exit* actions that are executed when the state is entered or left, respectively. A *transition* connects a *source* state to a *target* state. It may contain a *triggering event*, a *guard*, and an *effect*. A transition is taken when the triggering event is fired and the guard evaluates to true. When it is taken, the code representing the transition action is executed.

B. A Running Example

We use the control system of a simple traffic light as a running example. The top-level structure of the system is shown in Figure 1, which consists of three capsules: UserConsole *(UC)*, Controller *(CTR)*, and StopLightDriver *(SLD)*. The *CTR* is connected to *UC* and *SLD* using two ports, which are typed by interfaces ControlP and StopLightP accordingly. The *UC* component collects user input, which it passes on to the CTR component, the component controlling the light. Using the corresponding messages, the *CTR* component sends the control actions, to the *SLD* component which transfers the messages through a hardware port to the traffic light. The behaviour of *CTR* is shown in part 1 of the Fig. 4, which is intentionally left incomplete to demonstrate features of *Live-UMLRT*.

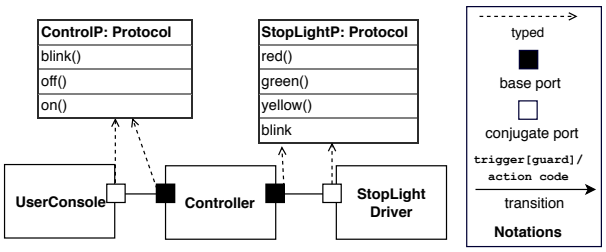
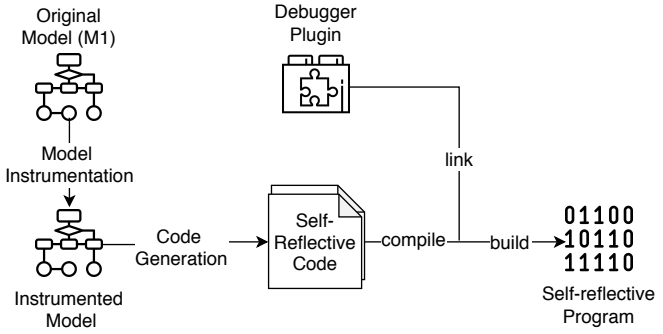
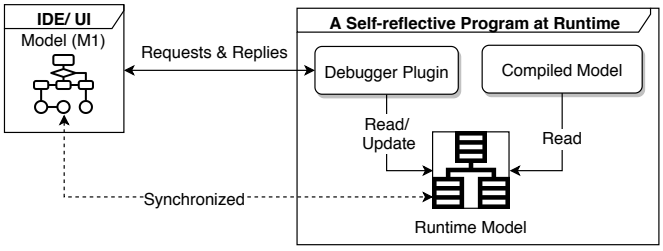


Fig. 1: Structure model of a simplified traffic light in UML-RT



(a) Generation of a self-reflective program.



(b) Live modeling using self-reflective program.

Fig. 2: An overview of our conceptual proposed framework.

III. Approach

A. An Overview

As discussed in Sec. I, the use of services offered in the context of live programming to implement live modeling imposes several challenges and restrictions. To overcome these, we propose a conceptual framework for live modeling in the context of model execution by code generation which is independent of live programming services. An overview of the framework is shown in Fig. 2. It consists of two phases: *Generation of a self-reflective program* and *Live modeling using the self-reflective program*. First, code generation and model transformation techniques are used to automatically create a program (self-reflective program) that embeds all required services for live modeling and debugging along with an interface for using them. Second, live modeling services are directly provided via interaction with the self-reflective program. In the following, we discuss the details of each phase in the context of UML-RT.

B. Instrumentation of UML-RT Models

Our approach adapts and extends the model instrumentation approach introduced in MDebugger [15], [16]. We use the variable view and change services from MDebugger and extend

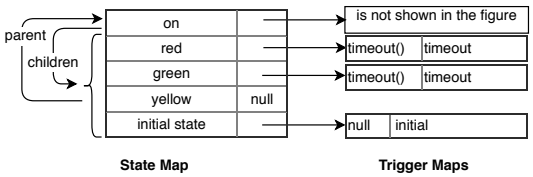


Fig. 3: State and transitions maps of the USM in part 1 of Figure 4.

them with a) support for the generation of execution traces required for state transfer and b) a live update mechanism that respects the execution semantics of the UML-RT model and prevents inconsistency.

C. Generation of the Self-reflective Program

The syntax for behavioural specification in UML-RT is concise and consists mainly of states, transitions, variables, and actions. In order to embed these elements in the generated code while still allowing for their modification at runtime, a runtime model is used. The most important part of the runtime model is a state map which is created per USM. A state map of an USM is a map from its states to references to trigger maps where a trigger map is a map from a trigger to a non-empty set of transitions. A trigger map of a state records the outgoing transitions from the state along with their trigger. A *null* trigger or a reference is used when a transition has no trigger, or a state has no outgoing transition. Figure 3 shows how the USM in part 1 of Figure 4 is translated to a state map and trigger maps referred to from the state map. At runtime, a state s is represented as references to its entry and exit actions (e.g., a function pointer in C++), the parent of s, and the childen of s (if any), as shown on the left of Figure 3. Similarly, for each transition references to its source and target states, guard and action are kept. The interested reader can refer to [17] and the source code of the implementation which has been made publicly available.

D. Supporting Live Modeling for UML-RT Using the Self-reflective Program

The live update services are defined based on the runtime model which is populated at the beginning of the execution by the self-reflective program. Any change in the design model is translated to a debugging command and sent to the debugger plugin which, in turn, applies it to the runtime model. The full range of edit operations on UML-RT models including adding/removing/updating states, transitions, actions, triggers, and variables (except remove) is provided. Most of the services are straight-forward to provide simply by adding, modifying, or removing entries in the runtime model. Interested readers can refer to [17] for a more detailed description.

IV. Live-UMLRT Features

In the following, we discuss the features of *Live-UMLRT* from the user point of view. When it is possible, the use of features is explained using the running example.

Setup and run: The Live-UMLRT is integrated into Papyrus-RT as an Eclipse plugin and can be downloaded and installed

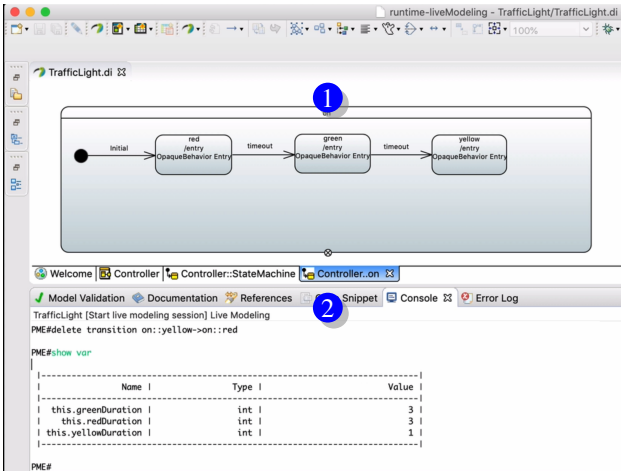


Fig. 4: User interface of Live-UMLRT

from the *Live-UMLRT* repository [18]. After installation, it can be used to edit UML-RT models at runtime simply by defining a run configuration (i.e., Eclipse run configuration) inside Papyrus-RT. When a model is executed using the defined configuration, first the model instrumentation, code generation, and build are executed automatically in the background without distracting end user, and a connection with debugger plugin is established. Upon a successful connection with the debugger plugin, a UI is loaded as shown in Figure 4. The UI is split in two parts, a *USM view* and a *DBG* console. In the USM view, the user can view and edit the USM of the capsules. In the *DBG* console the user can interactively issue commands to investigate and change the model at runtime. Basic debugging commands (e.g., view and change variables) are available and have been ported from MDebugger, our previous work on model-level debugging of UML-RT models [15], [16]. Next, we discuss the steps for live modeling along with several features of *Live-UMLRT*.

Starting live update session and applying changes: To start live modeling, a live update session must be started that moves the execution to a state in which the model execution can be changed consistently. A live update session can be started via two methods: 1) Interrupt the execution by pressing the 'b' key in the *DBG* console. This will stop the execution (similar to a debugging breakpoint) and allow users to apply changes into the model execution. This scenario is similar to the way that popular IDsE such as Eclipse support live programming work. 2) *Live-UMLRT* starts a live update session when the execution is stuck because of a missing specification in the model. The detection mechanism is ported from our previous work on partial model execution [19].

Applying changes to the model execution: During the live update session the user has two ways to update the model execution: **(1) Changing the design model:** the user can use *USM view* to update the model and save. During the save *Live-UMLRT* serializes the last change as update commands and sends them to the *debugger plugin*. As discussed, the debugger plugin applies changes by updating the runtime model. **(2) Changing the runtime model:** the user can issue

Listing 1: Supported Edit Operations

```
Add state: add state <name>
Delete/Update state: delete/update state <name>
Add transition: add transition (<name>)? <from>-><to
    > (when <signal> on <port>)?
Delete transition: delete transition (<name>)? <from
    >-><to>
Update transition: update transition (<name>)? <from
    >-><to> (when <signal> on <port>)?
Add action: record (action code)* save
Delete action: delete action (state|transition) (
    entry|exit)?
Add/Update variable: <name>=<expressions>
Replay execution: replay <state name>
```

the supported commands via the *DBG* console that validates and forwards the commands to the debugger plugin. With the current implementation, the changes from the runtime model are not propagated back to the design model. Thus, the changes affect the execution until the end of the live update session. Implementation of this part is left to future work.

Supported edit operations: Listing 1 shows the most important features supported by *Live-UMLRT*. In addition to the UI of *Live-UMLRT*, the features can be used via a TCP connection with the debugger plugin. In the following, we briefly discuss these features. (1) The add/delete/update commands are used to add/delete/update states and transitions. (2) A variable can be defined simply by setting an initial value to that. (3) To define a new action code, a record command should be used after which the UML-RT action code interpreter is activated. It accepts and interprets the action code line by line. Upon successful interpretation, the code can be saved to the runtime model as well as the design model. (4) Command *replay* allows the user to re-execute the previous execution steps to see the effect of the new changes. E.g., let us assume that the user completes the USM shown in part 1 by adding a transition from state *yellow* to state *red* when the execution is stuck in state *yellow* and unable to handle the received messages. Without a replay mechanism, there is no way to see the effect of the new change and the execution will be stuck in state *yellow* forever. However, by issuing `replay yellow`, *Live-UMLRT* steers the execution to state *yellow* again and injects the last messages that received in state *yellow*. By that, the execution can advance by processing the injected messages, and the user can see the effect of the new changes. A detailed description of the replay method can be found in [17].

V. Experimental Evaluation

This section details experiments we conducted to assess the performance and overhead of *Live-UMLRT*. For the experiments, several use cases are used. As shown in Table I, models have different complexities that range from eight states to more than 620 states. Models are described in [15].

Experiment 1 (Performance of creation of a self-reflective program). As discussed earlier, our approach for the creation of the self-reflective program consists of three steps: instrumentation of models, code generation, and compile/built. Creation of the self-reflective program is a core part of our approach. Thus we performed an experiment to measure the performance of the instrumentation of the model and code generation steps to evaluate the applicability of our approach. To do that, we ran our code generation, the model instrumentation, and the default code generation (i.e., the existing code generation of Papyrus-RT) 20 times against the use cases listed in Table I and in each case recorded the time required. **Results:** The *Code generation Time* and *Instrumentation Time* columns of Table I show the median of time required for instrumentation and code generation by our approach and the default code generation. For the largest model (Debuggable Failover), the median time of the instrumentation and code generation are less than 47 and 7 seconds, respectively. Code generation with the default code generator took 44 second which is slightly faster (3 seconds) than our approach. It is, therefore, safe to conclude that the code generation and instrumentation times of our approach is reasonable. Note that code generation and instrumentation are required only once for program generation and the required time appears negligible w.r.t. the benefits provided by the self-reflective program.

TABLE I: Model Complexity of Use-cases, Median of Code Generation and Instrumenting Time

Model	**Model Complexity**			**Code Gen. (ms)**		**Inst. (ms)**
	C	S	T	Def.	Ours	
Parcel Router	8	14	25	1140	1098	1277
Rover	6	16	21	1109	1187	1662
FailOver	7	31	43	1197	1274	2003
Debuggable FailOver	8	350	620	43623	46692	6200

C: Component, *S*: State, *T*: Transition, *Def.*: Default
Inst.: Instrumentation Time, *Code Gen.*: Code Generation Time

TABLE II: Performance of Edit Operation using Our Approach and Live Program

Operation	**One Edit (ms)**			**10 Edits, 3 Comps. (ms)**		
	Ours	Prog.	Ratio	Ours	Prog.	Ratio
Add State	1.3	608	467	10.6	1192	112
Rem./update State	1.5	608	405	11.5	1192	103
Add Trans.	1.5	608	405	12.9	1192	92
Rem./update Trans.	1.8	608	377	16.1	1192	74
Add Var	1.3	608	467	9.9	1192	120
Add Action	2.1	608	289	18.1	1192	66
Rem./update Action	1.5	608	405	12.3	1192	96
Average	**1.6**	**608**	**405**	**13**	**1192**	**92**

Ours: Our approach, *Prog.*: Live Programming
Trans.: Transition, *Ratio.*: Prog./Ours, *Rem.*: Remove

Experiment 2 (Performance of edit operation (Edit delay)). As discussed earlier, our approach provides live modeling via interaction with the self-reflective program and does not require code generation, compile/built, or hot-swapping for each edit. To show the efficiency of our approach, we measure the performance of edit operations using our approach and compare it with the approach relying on live programming. To do that, first, we executed the code generated by our approach, for the Debuggable Failover model and tried each of the edit operations 20 times using a random element and recorded the required times for each operation on a single element. Then we tried each edit operation on ten elements distributed over three different components and recorded the time needed for each

operation on these ten elements. Second, we repeated the same experiments on the code generated by an implementation of live modeling that relies on live programming. Also, since our approach interprets the actions as soon as they are modified during the live modeling, we executed action code with 100 lines of code in interpretation and compiled mode and recorded the CPU time in each case.

Results: Table II shows the median of the time required for edit operations using our approach and the approach relying on live programming. For a single operation on a single element and ten elements in three components, on average our approach is 400% and 92%, respectively, faster than when live programming services are used. As discussed, the main reason for this difference is the need for regeneration and recompilation after each change. Also, our experiment of the execution of actions in interpreted and compiled mode shows that, not surprisingly, the interpretation of actions is 70% slower than the execution of their compiled versions. Note that an action is only interpreted when it is edited during the live modeling. Also, our interpreter is a prototype, not built with performance optimization in mind, whereas C++ compilers are highly sophisticated and optimized.

In conclusion, our approach significantly improves the performance of edit operations (any change is applied in less than 2ms) which is considered acceptable in the context of live updates (e.g., according to [9], [10], users start noticing latency at 100ms and become distracted at 500ms). However, the execution of actions after the first edit deteriorated by 70%.

Experiment 3 (Memory and performance overhead of the generated code). Our approach generates the code that explicitly embeds the AST of the model. For that, we change the code generation which may affect the performance and memory usage of the created program. Thus, to show the performance and memory overhead of our approach, we executed the generated code of the Failover model in the context of our approach. During the execution we configured the system to process 10,000 client requests and recorded the CPU time and memory usage for processing the requests. Second, we repeated the same experiment using the code generated from the Failover model by the default code generation.

Results: The code generated from the Failover model using our approach took 514ms of CPU time to process 10,000 requests. This is only 1% slower than the time required by the code generated with the default code generator (504ms). Thus it is safe to conclude that the change in the generated code to support live modelling causes negligible performance overhead w.r.t. to the provided services. Also, the peak memory usage of code generated from the Failover model using our approach is 2083 KB to process 10,000 requests. This is 25% more than the memory usage by the code generated with the default code generator (1664 KB). We can argue this memory overhead is acceptable for many applications.

VI. Conclusion

In this paper, we have illustrated and validated *Live-UMLRT*, a tool that supports live modeling of UML-RT without relying on live programming services offered by the target language of the generated code. Our evaluation shows that *Live-UMLRT* (1) reduces the edit latency significantly, (2) is applicable with reasonable performance, (3) introduces negligible performance overhead and, (4) has an acceptable memory overhead for many application domains.

References

[1] F. Ciccozzi, I. Malavolta, and B. Selic, "Execution of UML models: a systematic review of research and practice," *Software & Systems Modeling*.

[2] S. Burckhardt, M. Fähndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato, "It's alive! Continuous feedback in UI programming," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 95–104.

[3] C. Schuster and C. Flanagan, "Live programming for event-based languages," in *Proceedings of the 2015 Reactive and Event-based Languages and Systems Workshop, REBLS*, vol. 15, 2015.

[4] S. McDirmid, "Usable live programming," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 2013, pp. 53–62.

[5] J. Kubelka, R. Robbes, and A. Bergel, "The road to live programming: insights from the practice," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1090–1101.

[6] R. van Rozen and T. van der Storm, "Toward live domain-specific languages," *Software & Systems Modeling*, vol. 18, no. 1, pp. 195–212, Feb 2019.

[7] U. Tikhonova, J. Stoel, T. Van Der Storm, and T. Degueule, "Constraint-based run-time state migration for live modeling," in *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2018, pp. 108–120.

[8] Y. VanTendeloo, S. VanMierlo, and H. Vangheluwe, "A multi-paradigm modelling approach to live modelling," *Software & Systems Modeling*, Oct 2018.

[9] S. McDirmid, "Living it up with a live programming language," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, ser. OOPSLA '07, 2007, pp. 623–638.

[10] ——, "Usable live programming," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2013, 2013, pp. 53–62.

[11] B. Selic, G. Gullekson, and P. T. Ward, *Real-time object-oriented modeling*. John Wiley & Sons New York, 1994, vol. 2.

[12] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, and D. Varró, "Survey and classification of model transformation tools," *Software & Systems Modeling*, Mar 2018. [Online]. Available: https://doi.org/10.1007/s10270-018-0665-6

[13] E. Posse and J. Dingel, "An Executable Formal Semantics for UML-RT," *Software & Systems Modeling*, vol. 15, no. 1, pp. 179–217, Feb. 2016.

[14] Eclipse Foundation, "Eclipse Papyrus for Real Time (Papyrus-RT)," https://www.eclipse.org/papyrus-rt, 2019, retrieved March 19, 2019.

[15] M. Bagherzadeh, N. Hili, and J. Dingel, "Model-level, platform-independent debugging in the context of the model-driven development of real-time systems," in *Proceedings of the 2017 25th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

[16] M. Bagherzadeh, N. Hili, D. Seekatz, and J. Dingel, "MDebugger: A Model-level Debugger for UML-RT," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18, 2018, pp. 97–100.

[17] M. Bagherzadeh, K. Jahed, B. Combemale, and J. Dingel, "Live modeling in the context of model-driven development and code generation," in *34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*, under review.

[18] M. Bagherzadeh, B. Jahed, Karim abd Combemale, and J. Dingel, "Live-UMLRT Repository," https://bitbucket.org/moji1/live-umlrt.git, 2019, retrieved July 19, 2019.

[19] M. Bagherzadeh, N. Kahani, K. Jahed, and J. Dingel, "Execution and debugging of partial models in the context of model-driven development," *IEEE Trans. Softw. Eng.*, under review.